# 6 Arrays

## WHAT'S IN THIS CHAPTER?

- Simple arrays
- Multidimensional arrays
- Jagged arrays
- The `Array` class
- Arrays as parameters
- Enumerators
- Spans
- Indices and ranges
- Array pools
- Bit arrays

## CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/Arrays`.

The code for this chapter is divided into the following major examples:

- SimpleArrays
- SortingSample
- YieldSample
- SpanSample

- IndicesAndRanges
- ArrayPoolSample
- BitArraySample

All the projects have nullable reference types enabled.

## MULTIPLE OBJECTS OF THE SAME TYPE

If you need to work with multiple objects of the same type, you can use collections (see Chapter 8, "Collections") and arrays. C# has a special notation to declare, initialize, and use arrays. Behind the scenes, the `Array` class comes into play, which offers several methods to sort and filter the elements inside the array. Using an enumerator, you can iterate through all the elements of the array.

> **NOTE** *For using multiple objects of different types, you can combine them using classes, structs, records, and tuples, which are covered in Chapter 3.*

## SIMPLE ARRAYS

If you need to use multiple objects of the same type, you can use an array. An *array* is a data structure that contains a number of elements of the same type.

### Array Declaration and Initialization

An array is declared by defining the type of elements inside the array, followed by empty brackets and a variable name. For example, an array containing integer elements is declared like this:

```
int[] myArray;
```

After declaring an array, memory must be allocated to hold all the elements of the array. An array is a reference type, so memory on the heap must be allocated. You do this by initializing the variable of the array using the `new` operator, with the type and the number of elements inside the array. Here, you specify the size of the array:

```
myArray = new int[4];
```

With this declaration and initialization, the variable `myArray` references four integer values that are allocated on the managed heap (see Figure 6-1).

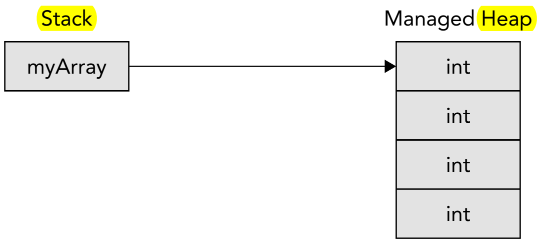

**FIGURE 6-1**

> **NOTE** *An array cannot be resized after its size is specified without copying all the elements. If you don't know how many elements should be in the array in advance, you can use a collection (see Chapter 8).*

Instead of using a separate line to declare and initialize an array, you can use a single line:

```
int[] myArray = new int[4];
```

You can also assign values to every array element using an array initializer. The following code samples all declare an array with the same content but with less code for you to write. The compiler can count the number of elements in the array by itself, which is why the array size is left out with the second line. The compiler also can map the values defined in the initializer list to the type used on the left side, so you also can remove the `new` operator left of the initializer. The code generated from the compiler is always the same:

```
int[] myArray1 = new int[4] {4, 7, 11, 2};
int[] myArray2 = new int[] {4, 7, 11, 2};
int[] myArray3 = {4, 7, 11, 2};
```

## Accessing Array Elements

After an array is declared and initialized, you can access the array elements using an indexer. Arrays support only indexers that have parameters of type `int`.

With the indexer, you pass the element number to access the array. The indexer always starts with a value of 0 for the first element. Therefore, the highest number you can pass to the indexer is the number of elements minus one because the index starts at zero. In the following example, the array `myArray` is declared and initialized with four integer values. The elements can be accessed with indexer values 0, 1, 2, and 3.

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0]; // read first element
int v2 = myArray[1]; // read second element
myArray[3] = 44; // change fourth element
```

**NOTE** *If you use a wrong indexer value that is bigger than the length of the array, an exception of type* `IndexOutOfRangeException` *is thrown.*

If you don't know the number of elements in the array, you can use the `Length` property, as shown in this `for` statement:

```
for (int i = 0; i < myArray.Length; i++)
{
  Console.WriteLine(myArray[i]);
}
```

Instead of using a `for` statement to iterate through all the elements of the array, you can also use the `foreach` statement:

```
foreach (var val in myArray)
{
  Console.WriteLine(val);
}
```

**NOTE** *The* `foreach` *statement makes use of the* `IEnumerable` *and* `IEnumerator` *interfaces and traverses through the array from the first index to the last. This is discussed in detail later in this chapter.*

## Using Reference Types

In addition to being able to declare arrays of predefined types, you also can declare arrays of custom types. Let's start with the following `Person` record using positional record syntax to declare the *init-only setter* properties `FirstName` and `LastName` (code file `SimpleArrays/Person.cs`):

```
public record Person(string FirstName, string LastName);
```

Declaring an array of two `Person` elements is similar to declaring an array of `int`:

```
Person[] myPersons = new Person[2];
```

However, be aware that if the elements in the array are reference types, memory must be allocated for every array element. If you use an item in the array for which no memory was allocated, a `NullReferenceException` is thrown.

**NOTE** *For information about errors and exceptions, see Chapter 10, "Errors and Exceptions."*

You can allocate every element of the array by using an indexer starting from 0. When you create the second object, you make use of C# 9 *target-typed new* as the type (code file `SimpleArrays/Program.cs`):

```
myPersons[0] = new Person("Ayrton", "Senna");
myPersons[1] = new("Michael", "Schumacher");
```

Figure 6-2 shows the objects in the managed heap with the `Person` array. `myPersons` is a variable that is stored on the stack. This variable references an array of `Person` elements that is stored on the managed heap. This array has enough space for two references. Every item in the array references a `Person` object that is also stored in the managed heap.

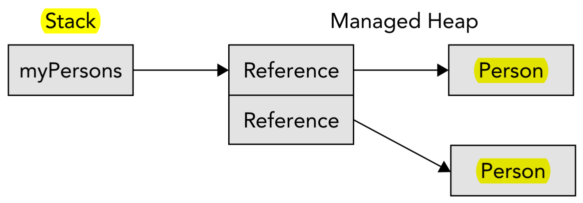

**FIGURE 6-2**

As with the `int` type, you can use an array initializer with custom types:

```
Person[] myPersons2 =
{
  new("Ayrton", "Senna"),
  new("Michael", "Schumacher")
};
```

## MULTIDIMENSIONAL ARRAYS

Ordinary arrays (also known as *one-dimensional arrays*) are indexed by a single integer. A multidimensional array is indexed by two or more integers.

$$a = \begin{bmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{bmatrix}$$

**FIGURE 6-3**

Figure 6-3 shows the mathematical notation for a two-dimensional array that has three rows and three columns. The first row has the values 1, 2, and 3, and the third row has the values 7, 8, and 9.

To declare this two-dimensional array with C#, you put a comma inside the brackets. The array is initialized by specifying the size of every dimension (also known as *rank*). Then the array elements can be accessed by using two integers with the indexer (code file `SimpleArrays/Program.cs`):

```
int[,] twodim = new int[3, 3];
twodim[0, 0] = 1;
twodim[0, 1] = 2;
twodim[0, 2] = 3;
twodim[1, 0] = 4;
twodim[1, 1] = 5;
twodim[1, 2] = 6;
twodim[2, 0] = 7;
twodim[2, 1] = 8;
twodim[2, 2] = 9;
```

**NOTE** *After declaring an array, you cannot change the rank.*

You can also initialize the two-dimensional array by using an array indexer if you know the values for the elements in advance. To initialize the array, one outer curly bracket is used, and every row is initialized by using curly brackets inside the outer curly brackets:

```
int[,] twodim = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

**NOTE** *When using an array initializer, you must initialize every element of the array. It is not possible to defer the initialization of some values until later.*

By using two commas inside the brackets, you can declare a three-dimensional array by placing initializers for two-dimensional arrays inside brackets separated by commas:

```
int[,,] threedim = {
  { { 1, 2 }, { 3, 4 } },
  { { 5, 6 }, { 7, 8 } },
  { { 9, 10 }, { 11, 12 } }
};
Console.WriteLine(threedim[0, 1, 1]);
```

Using a `foreach` loop, you can iterate through all the items of a multidimensional array.

## JAGGED ARRAYS

A two-dimensional array has a rectangular size (for example, 3 × 3 elements). A jagged array provides more flexibility in sizing the array. With a jagged array, every row can have a different size.

Figure 6-4 contrasts a two-dimensional array that has 3 × 3 elements with a jagged array. The jagged array shown contains three rows: the first row contains two elements, the second row contains six elements, and the third row contains three elements.

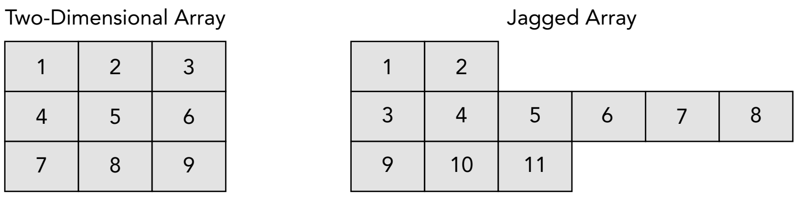

**FIGURE 6-4**

A jagged array is declared by placing one pair of opening and closing brackets after another. To initialize the jagged array, in the following code snippet an array initializer is used. The first array is initialized by items of arrays. Each of these items again is initialized with its own array initializer (code file `SimpleArrays/Program.cs`):

```
int[][] jagged =
{
  new[] { 1, 2 },
  new[] { 3, 4, 5, 6, 7, 8 },
  new[] { 9, 10, 11 }
};
```

You can iterate through all the elements of a jagged array with nested `for` loops. In the outer `for` loop, every row is iterated, and the inner `for` loop iterates through every element inside a row:

```
for (int row = 0; row < jagged.Length; row++)
{
  for (int element = 0; element < jagged[row].Length; element++)
  {
    Console.WriteLine($"row: {row}, element: {element}, " +
      $"value: {jagged[row][element]}");
  }
}
```

The output of the iteration displays the rows and every element within the rows:

```
row: 0, element: 0, value: 1
row: 0, element: 1, value: 2
row: 1, element: 0, value: 3
row: 1, element: 1, value: 4
row: 1, element: 2, value: 5
row: 1, element: 3, value: 6
row: 1, element: 4, value: 7
row: 1, element: 5, value: 8
row: 2, element: 0, value: 9
row: 2, element: 1, value: 10
row: 2, element: 2, value: 11
```

## ARRAY CLASS

Declaring an array with brackets is a C# notation using the `Array` class. Using the C# syntax behind the scenes creates a new class that derives from the abstract base class `Array`. This makes it possible to use methods and properties that are defined with the `Array` class with every C# array. For example, you've already used the `Length` property or iterated through the array by using the `foreach` statement. By doing this, you are using the `GetEnumerator` method of the `Array` class.

Other properties implemented by the `Array` class are `LongLength`, for arrays in which the number of items doesn't fit within an integer, and `Rank`, to get the number of dimensions.

Let's take a look at other members of the `Array` class by getting into various features.

### Creating Arrays

The `Array` class is abstract, so you cannot create an array by using a constructor. However, instead of using the C# syntax to create array instances, it is also possible to create arrays by using the static `CreateInstance` method. This is extremely useful if you don't know the type of elements in advance because the type can be passed to the `CreateInstance` method as a `Type` object.

The following example shows how to create an array of type `int` with a size of `5`. The first argument of the `CreateInstance` method requires the type of the elements, and the second argument defines the size. You can set values with the `SetValue` method and read values with the `GetValue` method (code file `SimpleArrays/Program.cs`):

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
  intArray1.SetValue(3 * i, i);
}

for (int i = 0; i < 5; i++)
{
  Console.WriteLine(intArray1.GetValue(i));
}
```

You can also cast the created array to an array declared as `int[]`:

```
int[] intArray2 = (int[])intArray1;
```

The `CreateInstance` method has many overloads to create multidimensional arrays and to create arrays that are not 0 based. The following example creates a two-dimensional array with 2 × 3 elements. The first dimension is 1 based; the second dimension is 10 based:

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

Setting the elements of the array, the `SetValue` method accepts indices for every dimension:

```
racers.SetValue(new Person("Alain", "Prost"), 1, 10);
racers.SetValue(new Person("Emerson", "Fittipaldi", 1, 11);
racers.SetValue(new Person("Ayrton", "Senna"), 1, 12);
racers.SetValue(new Person("Michael", "Schumacher"), 2, 10);
racers.SetValue(new Person("Fernando", "Alonso"), 2, 11);
racers.SetValue(new Person("Jenson", "Button"), 2, 12);
```

Although the array is not 0 based, you can assign it to a variable with the normal C# notation. You just have to take care not to cross the array boundaries:

```
Person[,] racers2 = (Person[,])racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

## Copying Arrays

Because arrays are reference types, assigning an array variable to another variable just gives you two variables referencing the same array. For copying arrays, the array implements the interface `ICloneable`. The `Clone` method that is defined with this interface creates a shallow copy of the array.

If the elements of the array are value types, as in the following code segment, all values are copied (see Figure 6-5):

```
int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();
```

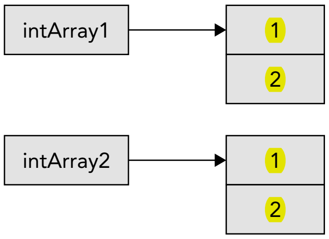

FIGURE 6-5

If the array contains reference types, only the references are copied, not the elements. Figure 6-6 shows the variables `beatles` and `beatlesClone`, where `beatlesClone` is created by calling the `Clone` method from `beatles`. The `Person` objects that are referenced are the same for `beatles` and `beatlesClone`. If you change a property of an element of `beatlesClone`, you change the same object of `beatles` (code file `SimpleArray/Program.cs`):

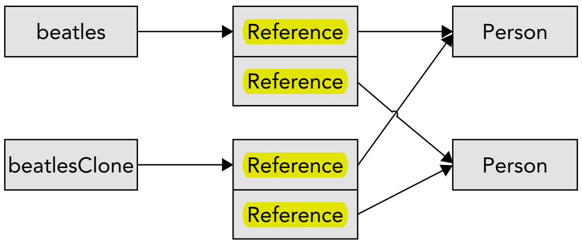

FIGURE 6-6

```
Person[] beatles = {
  new("John", "Lennon"),
  new("Paul", "McCartney")
};
Person[] beatlesClone = (Person[])beatles.Clone();
```

Instead of using the `Clone` method, you can use the `Array.Copy` method, which also creates a shallow copy. However, there's one important difference between `Clone` and `Copy`: `Clone` creates a new array; with `Copy` you have to pass an existing array with the same rank and enough elements.

> **NOTE** *If you need a deep copy of an array containing reference types, you have to iterate the array and create new objects.*

## Sorting

The `Array` class uses the Quicksort algorithm to sort the elements in the array. The `Sort` method requires the interface `IComparable` to be implemented by the elements in the array. Simple types such as `System.String` and `System.Int32` implement `IComparable`, so you can sort elements containing these types.

With the sample program, the array `names` contains elements of type `string`, and this array can be sorted (code file `SortingSample/Program.cs`):

```
string[] names = {
  "Lady Gaga",
  "Shakira",
  "Beyonce",
  "Ava Max"
};
Array.Sort(names);
foreach (var name in names)
{
  Console.WriteLine(name);
}
```

The output of the application shows the sorted result of the array:

```
Ava Max
Beyonce
Lady Gaga
Shakira
```

If you are using custom classes with the array, you must implement the interface `IComparable`. This interface defines just one method, `CompareTo`, which must return 0 if the objects to compare are equal; a value smaller than 0 if the instance should go before the object from the parameter; and a value larger than 0 if the instance should go after the object from the parameter.

Change the `Person` record to implement the interface `IComparable<Person>`. The comparison is first done on the value of the `LastName` by using the `Compare` method of the `String` class. If the `LastName` has the same value, the `FirstName` is compared (code file `SortingSample/Person.cs`):

```
public record Person(string FirstName, string LastName) : IComparable<Person>
{
  public int CompareTo(Person? other)
  {
    if (other == null) return 1;
    int result = string.Compare(this.LastName, other.LastName);
    if (result == 0)
    {
      result = string.Compare(this.FirstName, other.FirstName);
    }
    return result;
  }
  //...
```

Now it is possible to sort an array of `Person` objects by the last name (code file `SortingSample/Program.cs`):

```
Person[] persons = {
  new("Damon", "Hill"),
  new("Niki", "Lauda"),
  new("Ayrton", "Senna"),
  new("Graham", "Hill")
};

Array.Sort(persons);
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

Using `Array.Sort` with `Person` objects, the output returns the names sorted by last name:

```
Damon Hill
Graham Hill
Niki Lauda
Ayrton Senna
```

If the Person object should be sorted differently than the implementation within the Person class, a comparer type can implement the interface IComparer<T>. This interface specifies the method Compare, which defines two arguments that should be compared. The return value is similar to the result of the CompareTo method that's defined with the IComparable interface.

With the sample code, the class PersonComparer implements the IComparer<Person> interface to sort Person objects either by FirstName or by LastName. The enumeration PersonCompareType defines the different sorting options that are available with PersonComparer: FirstName and LastName. How the compare should be done is defined with the constructor of the class PersonComparer, where a PersonCompareType value is set. The Compare method is implemented with a switch statement to compare either by LastName or by FirstName (code file SortingSample/PersonComparer.cs):

```
public enum PersonCompareType
{
  FirstName,
  LastName
}

public class PersonComparer : IComparer<Person>
{
  private PersonCompareType _compareType;
  public PersonComparer(PersonCompareType compareType) =>
    _compareType = compareType;

  public int Compare(Person? x, Person? y)
  {
    if (x is null && y is null) return 0;
    if (x is null) return 1;
    if (y is null) return -1;

    return _compareType switch
    {
      PersonCompareType.FirstName => x.FirstName.CompareTo(y.FirstName),
      PersonCompareType.LastName => x.LastName.CompareTo(y.LastName),
      _ => throw new ArgumentException("unexpected compare type")
    };
  }
}
```

Now you can pass a PersonComparer object to the second argument of the Array.Sort method. Here, the people are sorted by first name (code file SortingSample/Program.cs):

```
Array.Sort(persons, new PersonComparer(PersonCompareType.FirstName));
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

The persons array is now sorted by first name:

```
Ayrton Senna
Damon Hill
Graham Hill
Niki Lauda
```

**NOTE** *The* `Array` *class also offers* `Sort` *methods that require a delegate as an argument. With this argument, you can pass a method to do the comparison of two objects rather than relying on the* `IComparable` *or* `IComparer` *interfaces. Chapter 7, "Delegates, Lambdas, and Events," discusses how to use delegates.*

## ARRAYS AS PARAMETERS

Arrays can be passed as parameters to methods and returned from methods. To return an array, you just have to declare the array as the return type, as shown with the following method `GetPersons`:

```
static Person[] GetPersons() =>
  new Person[] {
    new Person("Damon", "Hill"),
    new Person("Niki", "Lauda"),
    new Person("Ayrton", "Senna"),
    new Person("Graham", "Hill")
  };
```

When passing arrays to a method, the array is declared with the parameter, as shown with the method `DisplayPersons`:

```
static void DisplayPersons(Person[] persons)
{
  //...
}
```

## ENUMERATORS

By using the `foreach` statement, you can iterate elements of a collection (see Chapter 8) without needing to know the number of elements inside the collection. The `foreach` statement uses an enumerator. Figure 6-7 shows the relationship between the client invoking the `foreach` method and the collection. The array or collection implements the `IEnumerable` interface with the `GetEnumerator` method. The `GetEnumerator` method returns an enumerator implementing the `IEnumerator` interface. The interface `IEnumerator` is then used by the `foreach` statement to iterate through the collection.

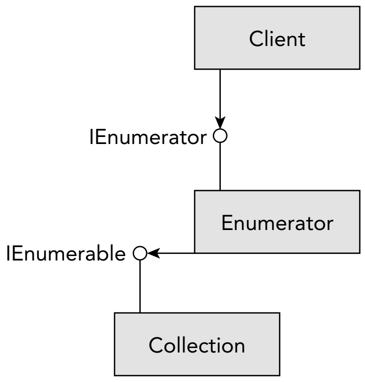

**FIGURE 6-7**

> **NOTE** *The* `GetEnumerator` *method is defined with the interface* `IEnumerable`. *The* `foreach` *statement doesn't really need this interface implemented in the collection class. It's enough to have a method with the name* `GetEnumerator` *that returns an object implementing the* `IEnumerator` *interface.*

## IEnumerator Interface

The `foreach` statement uses the methods and properties of the `IEnumerator` interface to iterate all elements in a collection. For this, `IEnumerator` defines the property `Current` to return the element where the cursor is positioned and defines the method `MoveNext` to move to the next element of the collection. `MoveNext` returns `true` if there's an element and `false` if no more elements are available.

The generic version of the interface `IEnumerator<T>` derives from the interface `IDisposable` and thus defines a `Dispose` method to clean up resources allocated by the enumerator.

> **NOTE** *The* `IEnumerator` *interface also defines the* `Reset` *method for COM interoperability. Many .NET enumerators implement this by throwing an exception of type* `NotSupported-Exception`.

## foreach Statement

The C# `foreach` statement is not resolved to a `foreach` statement in the IL code. Instead, the C# compiler converts the `foreach` statement to methods and properties of the `IEnumerator` interface. Here's a simple `foreach` statement to iterate all elements in the `persons` array and display them person by person:

```
foreach (var p in persons)
{
  Console.WriteLine(p);
}
```

The `foreach` statement is resolved to the following code fragment. First, the `GetEnumerator` method is invoked to get an enumerator for the array. Inside a `while` loop, as long as `MoveNext` returns `true`, the elements of the array are accessed using the `Current` property:

```
IEnumerator<Person> enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
  Person p = enumerator.Current;
  Console.WriteLine(p);
}
```

## yield Statement

Using the `foreach` statement, it's easy to use the `IEnumerable` and `IEnumerator` interfaces—the compiler converts the code to use the members of these interfaces. To create classes implementing these interfaces, the compiler offers the `yield` statement. When you use `yield return` and `yield break`, the compiler generates a state machine to iterate through a collection implementing the members of these interfaces. `yield return` returns one element of a collection and moves the position to the next element; `yield break` stops the iteration. The iteration also ends when the method is completed, so a `yield break` is only needed to stop earlier.

The next example shows the implementation of a simple collection using the `yield return` statement. The class `HelloCollection` contains the method `GetEnumerator`. The implementation of the `GetEnumerator` method contains two `yield return` statements where the strings `Hello` and `World` are returned (code file `YieldSample/Program.cs`):

```
class HelloCollection
{
  public IEnumerator<string> GetEnumerator()
  {
    yield return "Hello";
    yield return "World";
  }
}
```

> **NOTE** *A method or property that contains* `yield` *statements is also known as an iterator block. An iterator block must be declared to return an* `IEnumerator` *or* `IEnumerable` *interface or the generic versions of these interfaces. This block may contain multiple* `yield return` *or* `yield break` *statements; a* `return` *statement is not allowed.*

Now it is possible to iterate through the collection using a `foreach` statement:

```
public void HelloWorld()
{
  HelloCollection helloCollection = new();
  foreach (string s in helloCollection)
  {
    Console.WriteLine(s);
  }
}
```

> **NOTE** *Remember that the* `yield` *statement produces an enumerator and not just a list filled with items. This enumerator is invoked by the* `foreach` *statement. As each item is accessed from the* `foreach`*, the enumerator is accessed. This makes it possible to iterate through huge amounts of data without reading all the data into memory in one turn.*

## Different Ways to Iterate Through Collections

In a slightly larger and more realistic way than the Hello World example, you can use the `yield return` statement to iterate through a collection in different ways. The class `MusicTitles` enables iterating the titles in a default way with the `GetEnumerator` method, in reverse order with the `Reverse` method, and through a subset with the `Subset` method (code file `YieldSample/MusicTitles.cs`):

```
public class MusicTitles
{
  string[] names = {"Tubular Bells", "Hergest Ridge", "Ommadawn", "Platinum"};

  public IEnumerator<string> GetEnumerator()
  {
    for (int i = 0; i < 4; i++)
```

```
      {
        yield return names[i];
      }
    }

    public IEnumerable<string> Reverse()
    {
      for (int i = 3; i >= 0; i--)
      {
        yield return names[i];
      }
    }

    public IEnumerable<string> Subset(int index, int length)
    {
      for (int i = index; i < index + length; i++)
      {
        yield return names[i];
      }
    }
  }
```

**NOTE** *The default iteration supported by a class is the* `GetEnumerator` *method, which is defined to return* `IEnumerator`*. Named iterations return* `IEnumerable`*.*

The client code to iterate through the string array first uses the `GetEnumerator` method, which you don't have to write in your code because it is used by default with the implementation of the `foreach` statement. Then the titles are iterated in reverse, and finally a subset is iterated by passing the index and number of items to iterate to the `Subset` method (code file `YieldSample/Program.cs`):

```
MusicTitles titles = new();
foreach (var title in titles)
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("reverse");
foreach (var title in titles.Reverse())
{
  Console.WriteLine(title);
}
Console.WriteLine();

Console.WriteLine("subset");
foreach (var title in titles.Subset(2, 2))
{
  Console.WriteLine(title);
}
```

## USING SPAN WITH ARRAYS

For a fast way to access managed or unmanaged continuous memory, you can use the `Span<T>` struct. One example where `Span<T>` can be used is an array; the `Span<T>` struct holds continuous memory behind the scenes. Another example of a use for `Span<T>` is a long string.

Using `Span<T>`, you can directly access array elements. The elements of the array are not copied, but they can be used directly, which is faster than a copy.

In the following code snippet, first a simple `int` array is created and initialized. A `Span<int>` object is created, invoking the constructor and passing the array to the `Span<int>`. The `Span<T>` type offers an indexer, and thus the elements of the `Span<T>` can be accessed using this indexer. Here, the second element is changed to the value `11`. Because the array `arr1` is referenced from the span, the second element of the array is changed by changing the `Span<T>` element. Finally, the span is returned from this method because it is used within top-level statements to pass it on to the next methods that follow (code file `SpanSample/Program.cs`):

```
Span<int> IntroSpans()
{
  int[] arr1 = { 1, 4, 5, 11, 13, 18 };
  Span<int> span1 = new(arr1);
  span1[1] = 11;
  Console.WriteLine($"arr1[1] is changed via span1[1]: {arr1[1]}");
  return span1;
}
```

### Creating Slices

A powerful feature of `Span<T>` is that you can use it to access parts, or *slices*, of an array. By using the slices, the array elements are not copied; they're directly accessed from the span.

The following code snippet shows two ways to create slices. With the first one, a constructor overload is used to pass the start and length of the array that should be used. With the variable `span3` that references this newly created `Span<int>`, it's only possible to access three elements of the array `arr2`, starting with the fourth element. Another overload of the constructor exists where you can pass just the start of the slice. With this overload, the remains of the array are taken until the end. You can also create a slice from a `Span<T>` object, invoking the `Slice` method. Similar overloads exist here. With the variable `span4`, the previously created `span1` is used to create a slice with four elements starting with the third element of `span1` (code file `SpanSample/Program.cs`):

```
private static Span<int> CreateSlices(Span<int> span1)
{
  Console.WriteLine(nameof(CreateSlices));
  int[] arr2 = { 3, 5, 7, 9, 11, 13, 15 };
  Span<int> span2 = new(arr2);
  Span<int> span3 = new(arr2, start: 3, length: 3);
  Span<int> span4 = span1.Slice(start: 2, length: 4);

  DisplaySpan("content of span3", span3);
  DisplaySpan("content of span4", span4);
  Console.WriteLine();
  return span2;
}
```

You use the `DisplaySpan` method to display the contents of a span. The following code snippet makes use of the `ReadOnlySpan`. You can use this span type if you don't need to change the content that the span references, which is the case in the `DisplaySpan` method. `ReadOnlySpan<T>` is discussed later in this chapter in more detail:

```
private static void DisplaySpan(string title, ReadOnlySpan<int> span)
{
  Console.WriteLine(title);
  for (int i = 0; i < span.Length; i++)
  {
    Console.Write($"{span[i]}.");
  }
  Console.WriteLine();
}
```

When you run the application, the content of `span3` and `span4` is shown—a subset of the `arr2` and `arr1`:

```
content of span3
9.11.13.
content of span4
6.8.10.12.
```

**NOTE** `Span<T>` *is safe from crossing the boundaries. If you're creating spans that exceed the contained array length, an exception of type* `ArgumentOutOfRangeException` *is thrown. Read Chapter 10 for more information on exception handling.*

## Changing Values Using Spans

You've seen how to directly change elements of the array that are referenced by the span using the indexer of the `Span<T>` type. There are more options as shown in the following code snippet.

You can invoke the `Clear` method, which fills a span containing `int` types with `0`; you can invoke the `Fill` method to fill the span with the value passed to the `Fill` method; and you can copy a `Span<T>` to another `Span<T>`. With the `CopyTo` method, if the destination span is not large enough, an exception of type `ArgumentException` is thrown. You can avoid this outcome by using the `TryCopyTo` method. This method doesn't throw an exception if the destination span is not large enough; instead, it returns `false` as being not successful with the copy (code file `SpanSample/Program.cs`):

```
private static void ChangeValues(Span<int> span1, Span<int> span2)
{
  Console.WriteLine(nameof(ChangeValues));
  Span<int> span4 = span1.Slice(start: 4);
  span4.Clear();
  DisplaySpan("content of span1", span1);
  Span<int> span5 = span2.Slice(start: 3, length: 3);
  span5.Fill(42);
  DisplaySpan("content of span2", span2);
  span5.CopyTo(span1);
  DisplaySpan("content of span1", span1);
```

```
    if (!span1.TryCopyTo(span4))
    {
      Console.WriteLine("Couldn't copy span1 to span4 because span4 is " +
        "too small");
      Console.WriteLine($"length of span4: {span4.Length}, length of " +
        $"span1: {span1.Length}");
    }
    Console.WriteLine();
  }
```

When you run the application, you can see the content of `span1` where the last two numbers have been cleared using `span4`, the content of `span2` where `span5` was used to fill the value `42` with three elements, and again the content of `span1` where the first three numbers have been copied over from `span5`. Copying `span1` to `span4` was not successful because `span4` has just a length of 4, whereas `span1` has a length of 6:

```
content of span1
2.11.6.8.0.0.
content of span2
3.5.7.42.42.42.15.
content of span1
42.42.42.8.0.0.
Couldn't copy span1 to span4 because span4 is too small
length of span4: 2, length of span1: 6
```

## ReadOnly Spans

If you need only read-access to an array segment, you can use `ReadOnlySpan<T>` as was already shown in the `DisplaySpan` method. With `ReadOnlySpan<T>`, the indexer is read-only, and this type doesn't offer `Clear` and `Fill` methods. You can, however, invoke the `CopyTo` method to copy the content of the `ReadOnlySpan<T>` to a `Span<T>`.

The following code snippet creates `readOnlySpan1` from an array with the constructor of `ReadOnlySpan<T>`. `readOnlySpan2` and `readOnlySpan3` are created by direct assignments from `Span<int>` and `int[]`. Implicit cast operators are available with `ReadOnlySpan<T>` (code file `SpanSample/Program.cs`):

```
void ReadonlySpan(Span<int> span1)
{
  Console.WriteLine(nameof(ReadonlySpan));
  int[] arr = span1.ToArray();
  ReadOnlySpan<int> readOnlySpan1 = new(arr);
  DisplaySpan("readOnlySpan1", readOnlySpan1);

  ReadOnlySpan<int> readOnlySpan2 = span1;
  DisplaySpan("readOnlySpan2", readOnlySpan2);
  ReadOnlySpan<int> readOnlySpan3 = arr;
  DisplaySpan("readOnlySpan3", readOnlySpan3);
  Console.WriteLine();
}
```

**NOTE** *How to implement implicit cast operators is discussed in Chapter 5, "Operators and Casts." Read more information on spans in Chapter 13, "Managed and Unmanaged Memory."*

**NOTE** *Previous editions of this book demonstrated the use of* `ArraySegment<T>`*. Although* `ArraySegment<T>` *is still available, it has some shortcomings, and you can use the more flexible* `Span<T>` *as a replacement. In case you're already using* `ArraySegment<T>`*, you can keep the code and interact with spans. The constructor of* `Span<T>` *also allows passing an* `ArraySegment<T>` *to create a* `Span<T>` *instance.*

## INDICES AND RANGES

Starting with C# 8, indices and ranges based on the `Index` and `Range` types were included, along with the range and hat operators. Using the hat operator, you can access elements counting from the end.

### Indices and the Hat Operator

Let's start with the following array, which consists of nine integer values (code file `IndicesAndRanges/Program.cs`):

```
int[] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The traditional way to access the first and last elements of this array is to use the indexer implemented with the `Array` class, pass an integer value for the nth element starting with 0 for the first element, and use the length minus 1 for the last element:

```
int first1 = data[0];
int last1 = data[data.Length - 1];
Console.WriteLine($"first: {first1}, last: {last1}");
```

With the hat operator (`^`), you can use `^1` to access the last element, and the calculation based on the length is no longer necessary:

```
int last2 = data[^1];
Console.WriteLine(last2);
```

Behind the scenes, the `Index` struct type is used. An implicit cast from `int` to `Index` is implemented, so you can assign `int` values to the `Index` type. Using the hat operator, the compiler creates an `Index` that initializes the `IsFromEnd` property to `true`. Passing the `Index` to an indexer, the compiler converts the value to an `int`. If the `Index` starts from the end, calculation is done with either a `Length` or a `Count` property (depending on what property is available):

```
Index firstIndex = 0;
Index lastIndex = ^1;
int first3 = data[firstIndex];
int last3 = data[lastIndex];
Console.WriteLine($"first: {first3}, last: {last3}");
```

### Ranges

To access a range of the array, the range operator (`..`) can be used with the underlying `Range` type. In the sample code, the `ShowRange` method is implemented to display the values of an array with a string output (code file `IndicesAndRanges/Program.cs`):

```
void ShowRange(string title, int[] data)
{
  Console.WriteLine(title);
  Console.WriteLine(string.Join(" ", data));
  Console.WriteLine();
}
```

By invoking this method with different values passed using the range operator, you can see the various forms of ranges. A range is defined with `..` embedded with an `Index` on the left and an `Index` on the right. Starting with `..` and omitting the `Index` from the left side just starts from the beginning. Omitting the `Index` from the right side, the range goes up to the end. Using `..` with the array just returns the complete array.

The `Index` on the left side specifies an inclusive value, whereas the `Index` on the right side is exclusive. With the end of the range, you need to specify the element following the last element you want to access. When the `Index` type was used before, you've seen that `^1` references the last value of the collection. When using the `Index` on the right side of a range, you must specify `^0` to address the element after the last element (remember the right side of the range is exclusive).

With the code sample, a full range is used (`..`), the first three elements are passed with `0..3`; the fourth to the sixth elements are passed with `3..6`; and counting from the end, the last three elements are passed with `^3..^0`:

```
ShowRange("full range", data[..]);
ShowRange("first three", data[0..3]);
ShowRange("fourth to sixth", data[3..6]);
ShowRange("last three", data[^3..^0]);
```

Behind the scenes, the `Range` struct type is used, and you can assign ranges to variables:

```
Range fullRange = ..;
Range firstThree = 0..3;
Range fourthToSixth = 3..6;
Range lastThree = ^3..^0;
```

The `Range` type specifies a constructor that passes two `Index` values for the start and the end, `End` and `Start` properties that return an `Index`, and a `GetOffsetAndLength` method that returns a tuple consisting of the offset and length of a range.

## Efficiently Changing Array Content

Using a range of an array, the array elements are copied. Changing values within the range, the original values of the array do not change. However, as described in the section "Using Span with Arrays," a `Span` allows accessing a slice of an array directly. The `Span` type also supports indices and ranges, and you can change the content of an array by accessing a range of the `Span` type.

The following code snippet demonstrates accessing a slice of an array and changing the first element of the slice; the original value of the array didn't change because a copy was done. In the code lines that follow, a `Span` is created to access the array using the `AsSpan` method. With this `Span`, the range operator is used, which in turn invokes the `Slice` method of the `Span`. Changing values from this slice, the array is directly accessed and changed using an indexer on the slice (code file `IndicesAndRanges/Program.cs`):

```
var slice1 = data[3..5];
slice1[0] = 42;
Console.WriteLine($"value in array didn't change: {data[3]}, " +
  $"value from slice: {slice1[0]}");
```

```
var sliceToSpan = data.AsSpan()[3..5];
sliceToSpan[0] = 42;
Console.WriteLine($"value in array: {data[3]}, value from slice: {sliceToSpan[0]}");
```

## Indices and Ranges with Custom Collections

To support indices and ranges with custom collections, not a lot of work is required. To support the hat operator, the `MyCollection` class implements an indexer and the `Length` property. To support ranges, you can either create a method that receives a `Range` type or—a simpler way—create a method with the name `Slice` that has two `int` parameters and can have the return type you need. The compiler converts the range to calculate the start and length (code file `IndicesAndRanges/MyCollection.cs`):

```
using System;
using System.Linq;

public class MyCollection
{
  private int[] _array = Enumerable.Range(1, 100).ToArray();

  public int Length => _array.Length;

  public int this[int index]
  {
    get => _array[index];
    set => _array[index] = value;
  }

  public int[] Slice(int start, int length)
  {
    var slice = new int[length];
    Array.Copy(_array, start, slice, 0, length);
    return slice;
  }
}
```

The collection is initialized. With just the few lines that have been implemented, the hat operator can be used with the indexer, and with the range operator, the compiler converts this to invoke the `Slice` method (code file `IndicesAndRanges/Program.cs`):

```
MyCollection coll = new();
int n = coll[^20];
Console.WriteLine($"Item from the collection: {n}");
ShowRange("Using custom collection", coll[45..^40]);
```

## ARRAY POOLS

If you have an application where a lot of arrays are created and destroyed, the garbage collector has some work to do. To reduce the work of the garbage collector, you can use array pools with the `ArrayPool` class (from the namespace `System.Buffers`). `ArrayPool` manages a pool of arrays. Arrays can be rented from and returned to the pool. Memory is managed from the `ArrayPool` itself.

## Creating the Array Pool

You can create an `ArrayPool<T>` by invoking the static `Create` method. For efficiency, the array pool manages memory in multiple buckets for arrays of similar sizes. With the `Create` method, you can define the maximum array length and the number of arrays within a bucket before another bucket is required:

```
ArrayPool<int> customPool = ArrayPool<int>.Create(
  maxArrayLength: 40000, maxArraysPerBucket: 10);
```

The default for the `maxArrayLength` is 1024 × 1024 bytes, and the default for `maxArraysPerBucket` is 50. The array pool uses multiple buckets for faster access to arrays when many arrays are used. Arrays of similar sizes are kept in the same bucket as long as possible, and the maximum number of arrays is not reached.

You can also use a predefined shared pool by accessing the `Shared` property of the `ArrayPool<T>` class:

```
ArrayPool<int> sharedPool = ArrayPool<int>.Shared;
```

## Renting Memory from the Pool

Requesting memory from the pool happens by invoking the `Rent` method. The `Rent` method accepts the minimum array length that should be requested. If memory is already available in the pool, it is returned. If it is not available, memory is allocated for the pool and returned afterward. In the
following code snippet, an array of 1024, 2048, 3096, and so on elements is requested in a `for` loop (code file `ArrayPoolSample/Program.cs`):

```
private static void UseSharedPool()
{
  for (int i = 0; i < 10; i++)
  {
    int arrayLength = (i + 1) << 10;
    int[] arr = ArrayPool<int>.Shared.Rent(arrayLength);
    Console.WriteLine($"requested an array of {arrayLength} " +
      $"and received {arr.Length}");
    //...
  }
}
```

The `Rent` method returns an array with at least the requested number of elements. The array returned could have more memory available. The shared pool keeps arrays with at least 16 elements. The element count of the managed arrays always doubles—for example, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 elements, and so on.

When you run the application, you can see that larger arrays are returned if the requested array size doesn't fit the arrays managed by the pool:

```
requested an array of 1024 and received 1024
requested an array of 2048 and received 2048
requested an array of 3072 and received 4096
requested an array of 4096 and received 4096
requested an array of 5120 and received 8192
requested an array of 6144 and received 8192
requested an array of 7168 and received 8192
requested an array of 8192 and received 8192
requested an array of 9216 and received 16384
requested an array of 10240 and received 16384
```

## Returning Memory to the Pool

After you no longer need the array, you can return it to the pool. After the array is returned, you can later reuse it by renting it again.

You return the array to the pool by invoking the `Return` method of the array pool and passing the array to the `Return` method. With an optional parameter, you can specify whether the array should be cleared before it is returned to the pool. Without clearing it, the next one renting an array from the pool could read the data. By clearing the data, you avoid this, but you need more CPU time (code file: `ArrayPoolSample/Program.cs`):

```
ArrayPool<int>.Shared.Return(arr, clearArray: true);
```

**NOTE** *Information about the garbage collector and how to get information about memory addresses is in Chapter 13.*

## BITARRAY

If you need to work with an array of bits, you can use the `BitArray` type (from the namespace `System.Collections`). `BitArray` is a reference type that contains an array of `int`s, where for every 32 bits a new integer is used. `BitArray` defines `Count` and `Length` properties, an indexer, a `SetAll` method to set all the bits according to the parameters passed, a `Not` method to inverse the bits, as well as `And`, `Or`, and `Xor` methods for binary AND, binary OR, and exclusive OR.

**NOTE** *Chapter 5 covers bitwise operators that can be used with number types such as* `byte`, `short`, `int`, *and* `long`. *The* `BitArray` *class has similar functionality but can be used with a different number of bits than the C# types.*

With the code sample, the extension method `GetBitsFormat` iterates through a `BitArray` and writes `1` or `0` to a `StringBuilder`, depending on whether the bit is set. For better readability, a separator character is added every four bits (code file `BitArraySample/BitArrayExtensions.cs`):

```
public static class BitArrayExtensions
{
  public static string GetBitsFormat(this BitArray bits)
  {
    StringBuilder sb = new();
    for (int i = bits.Length - 1; i >= 0; i--)
    {
      sb.Append(bits[i] ? 1 : 0);
      if (i != 0 && i % 4 == 0)
      {
        sb.Append("_");
      }
    }
    return sb.ToString();
  }
}
```

The following example demonstrates the `BitArray` class creating a bit array with nine bits, indexed from 0 to 8. The `SetAll` method sets all nine bits to `true`. Then the `Set` method changes bit `1` to `false`. Instead of the `Set` method, you can also use an indexer, as shown with index `5` and `7` (code file `BitArraySample/Program.cs`):

```
BitArray bits1 = new(9);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.Write("initialized: ");
Console.WriteLine(bits1.GetBitsFormat());
Console.WriteLine();
```

This is the displayed result of the initialized bits:

```
initialized: 1_0101_1101
```

The `Not` method generates the inverse of the bits of the `BitArray`:

```
Console.WriteLine($"NOT {bits1.FormatString()}");
bits1.Not();
Console.WriteLine($" =  {bits1.FormatString()}");
Console.WriteLine();
```

The result of `Not` is all bits inverted. If the bit were `true`, it is `false`; and if it were `false`, it is `true`:

```
NOT 1_0101_1101
 =  0_1010_0010
```

In the following example, a new `BitArray` is created. With the constructor, the variable `bits1` is used to initialize the array, so the new array has the same values. Then the values for bits 0, 1, and 4 are set to different values. Before the `Or` method is used, the bit arrays `bits1` and `bits2` are displayed. The `Or` method changes the values of `bits1`:

```
BitArray bits2 = new(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
Console.WriteLine($"   {bits1.FormatString()}");
Console.WriteLine($"OR {bits2.FormatString()}");
bits1.Or(bits2);
Console.WriteLine($"=  {bits1.FormatString()}");
Console.WriteLine();
```

With the `Or` method, the set bits are taken from both input arrays. In the result, the bit is set if it was set with either the first or the second array:

```
   0_1010_0010
OR 0_1011_0001
=  0_1011_0011
```

Next, the `And` method is used to operate on `bits2` and `bits1`:

```
Console.WriteLine($"    {bits2.FormatString()}");
Console.WriteLine($"AND {bits1.FormatString()}");
bits2.And(bits1);
Console.WriteLine($"=   {bits2.FormatString()}");
Console.WriteLine();
```

The result of the `And` method only sets the bits where the bit was set in both input arrays:

```
    0_1011_0001
AND 0_1011_0011
=   0_1011_0001
```

Finally, the `Xor` method is used for an exclusive `OR`:

```
Console.WriteLine($"    {bits1.FormatString()} ");
Console.WriteLine($"XOR {bits2.FormatString()}");
bits1.Xor(bits2);
Console.WriteLine($"=   {bits1.FormatString()}");
Console.ReadLine();
```

With the `Xor` method, the resultant bit is set only if the bit was set either in the first or second input, but not both:

```
    0_1011_0011
XOR 0_1011_0001
=   0_0000_0010
```

## SUMMARY

This chapter covered how to use the C# notation to create and use simple, multidimensional, and jagged arrays. The `Array` class is used behind the scenes of C# arrays, enabling you to invoke properties and methods of this class with array variables.

You saw how to sort elements in the array by using the `IComparable` and `IComparer` interfaces; and you learned how to create and use enumerators, the interfaces `IEnumerable` and `IEnumerator`, and the `yield` statement.

With the `Span<T>` type, you saw efficient ways to access a slice of the array. You also saw range and index enhancements with C#.

The last sections of this chapter showed you how to efficiently use arrays with the `ArrayPool`, as well as how to use the BitArray type to deal with an array of bits.

The next chapter gets into details of more important features of C#: delegates, lambdas, and events.